Digital Design & Computer Arch.

Lecture 21b: Memory Hierarchy and Caches

Prof. Onur Mutlu

ETH Zürich

Spring 2020

14 May 2020

Readings for Today
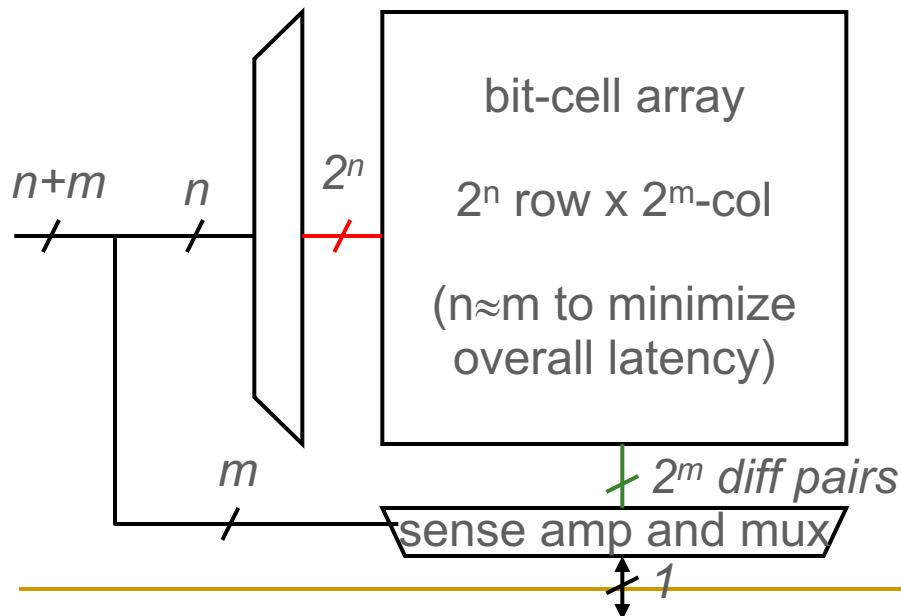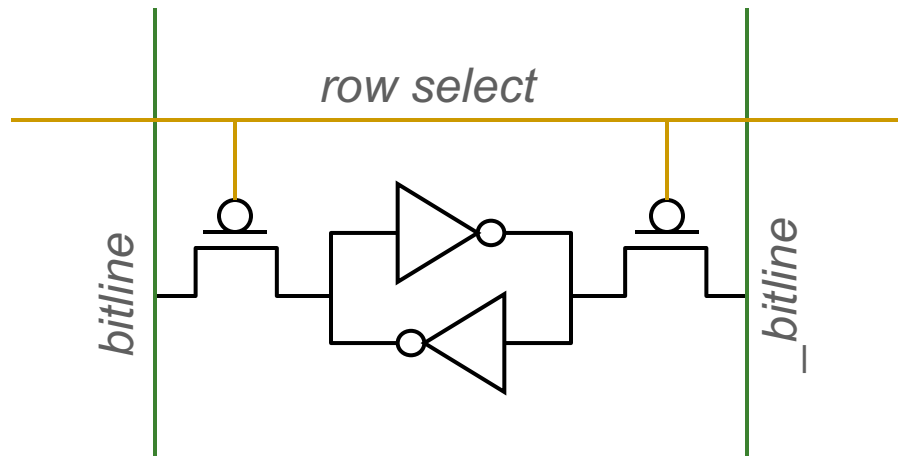
- **Memory Hierarchy and Caches**

- Required
  - ❑ H&H Chapters 8.1-8.3
  - ❑ Refresh: P&P Chapter 3.5

- Recommended
  - ❑ An early cache paper by Maurice Wilkes
    - ▪ Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
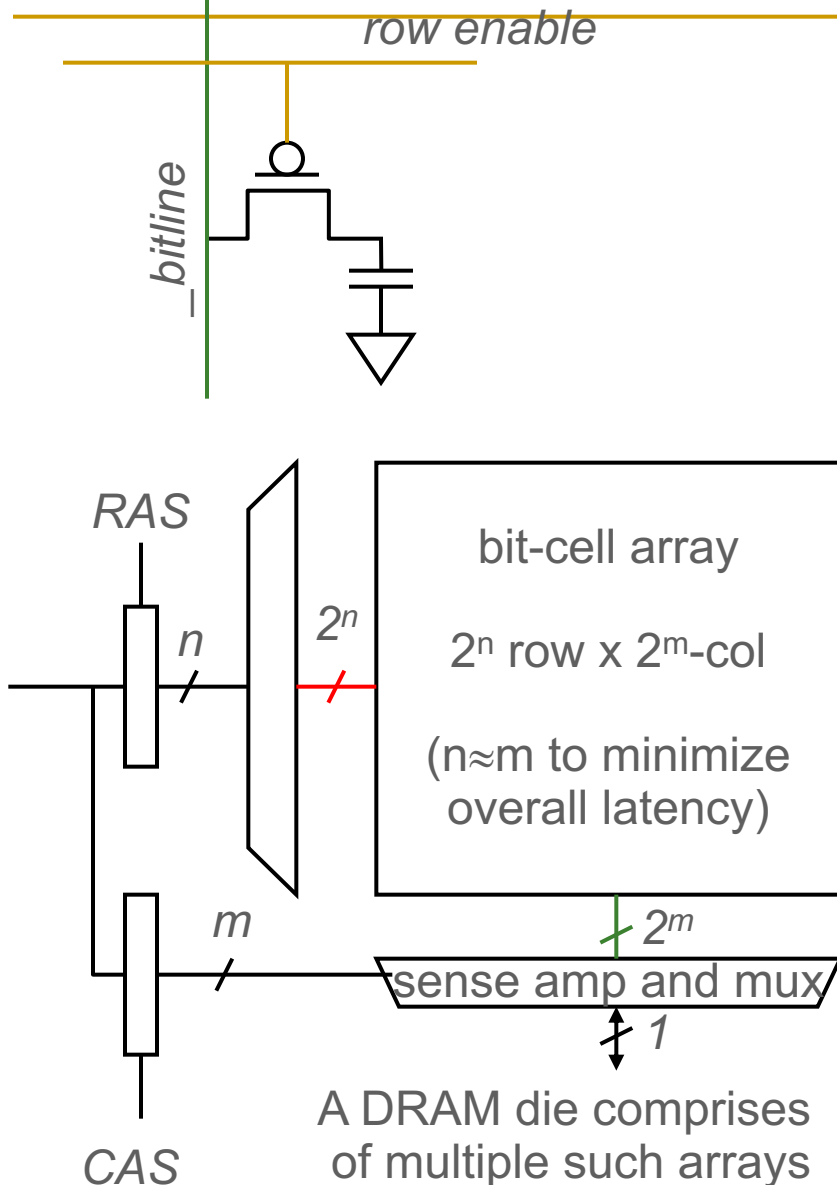
Recall: SRAM

Read Sequence

1. address decode
2. drive row select
3. selected bit-cells drive bitlines
    (entire row is read together)
4. differential sensing and column select
    (data is ready)
5. precharge all bitlines
    (for next read or write)

Access latency dominated by steps 2 and 3

Cycling time dominated by steps 2, 3 and 5

- step 2 proportional to 2^m
- step 3 and 5 proportional to 2^n

3

Recall: DRAM

row enable

bitline

RAS

n

2^n

m

CAS

bit-cell array

2^n row x 2^m-col

(n≈m to minimize overall latency)

2^m

sense amp and mux

1

A DRAM die comprises of multiple such arrays

Bits stored as charges on node capacitance (non-restorative)

- bit cell loses charge when read
- bit cell loses charge over time

Read Sequence

1~3 same as SRAM

4. a "flip-flopping" sense amp amplifies and regenerates the bitline, data bit is mux'ed out

5. precharge all bitlines

Destructive reads

Charge loss over time

Refresh: A DRAM controller must periodically read each row within the allowed refresh time (10s of ms) such that charge is restored

DRAM vs. SRAM

- DRAM
  - Slower access (capacitor)
  - Higher density (1T 1C cell)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together

- SRAM
  - Faster access (no capacitor)
  - Lower density (6T cell)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)

The Memory Hierarchy

Memory in a Modern System
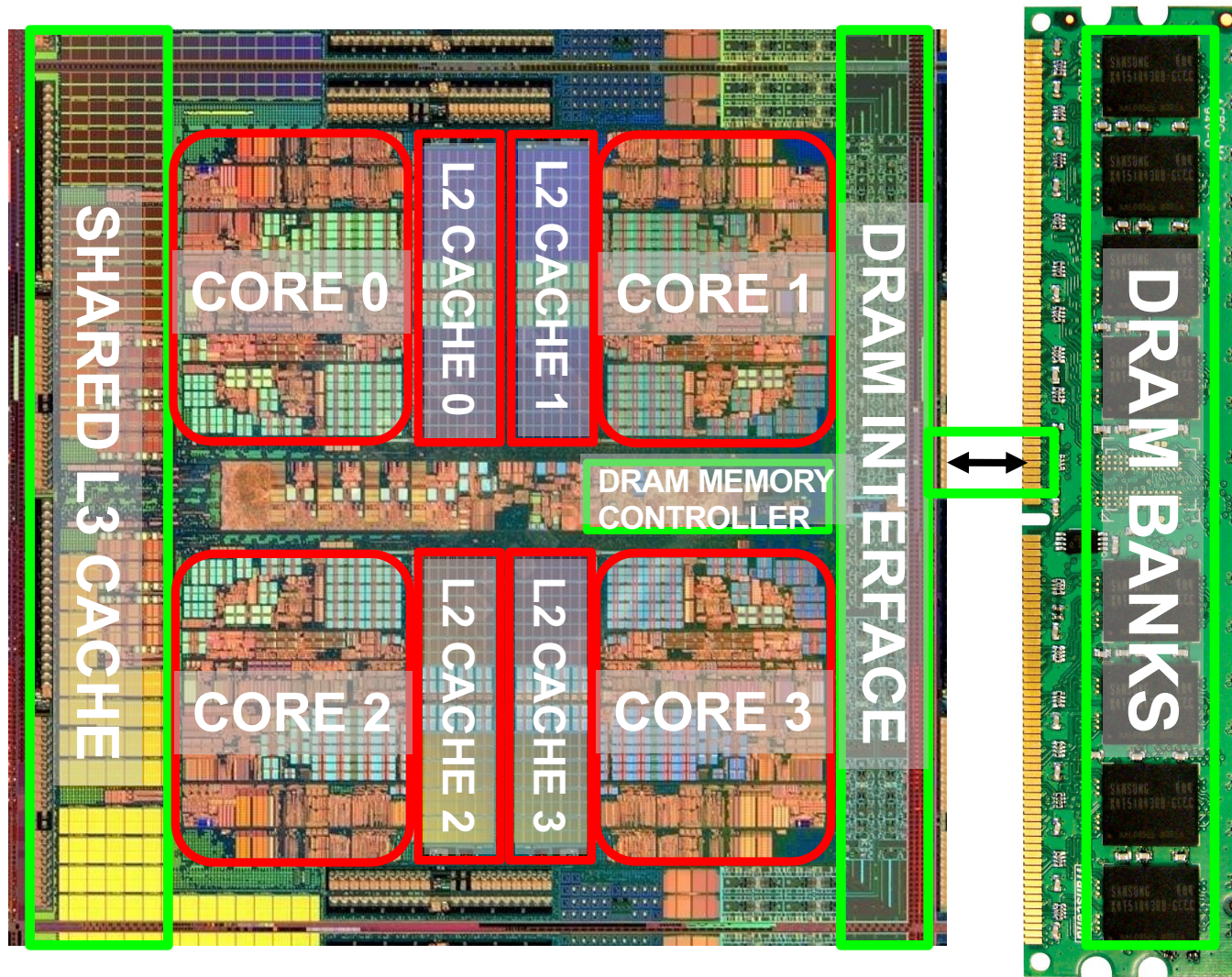
Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)

The Problem

- Ideal memory's requirements oppose each other

- Bigger is slower
  - Bigger → Takes longer to determine the location

- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. Disk vs. Tape

- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology

The Problem

- **Bigger is slower**
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec

- **Faster is more expensive (dollars and chip area)**
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
  - These sample values (circa ~2011) scale with time

- Other technologies have their place as well
  - Flash memory (mature), PC-RAM, MRAM, RRAM (not mature yet)

Why Memory Hierarchy?

- We want both fast and large

- But we cannot achieve both with a single level of memory

- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)

The Memory Hierarchy

move what you use here

fast
small

With good locality of reference, memory appears as fast as and as large as

backup everything here

big but slow

faster per byte

cheaper per byte
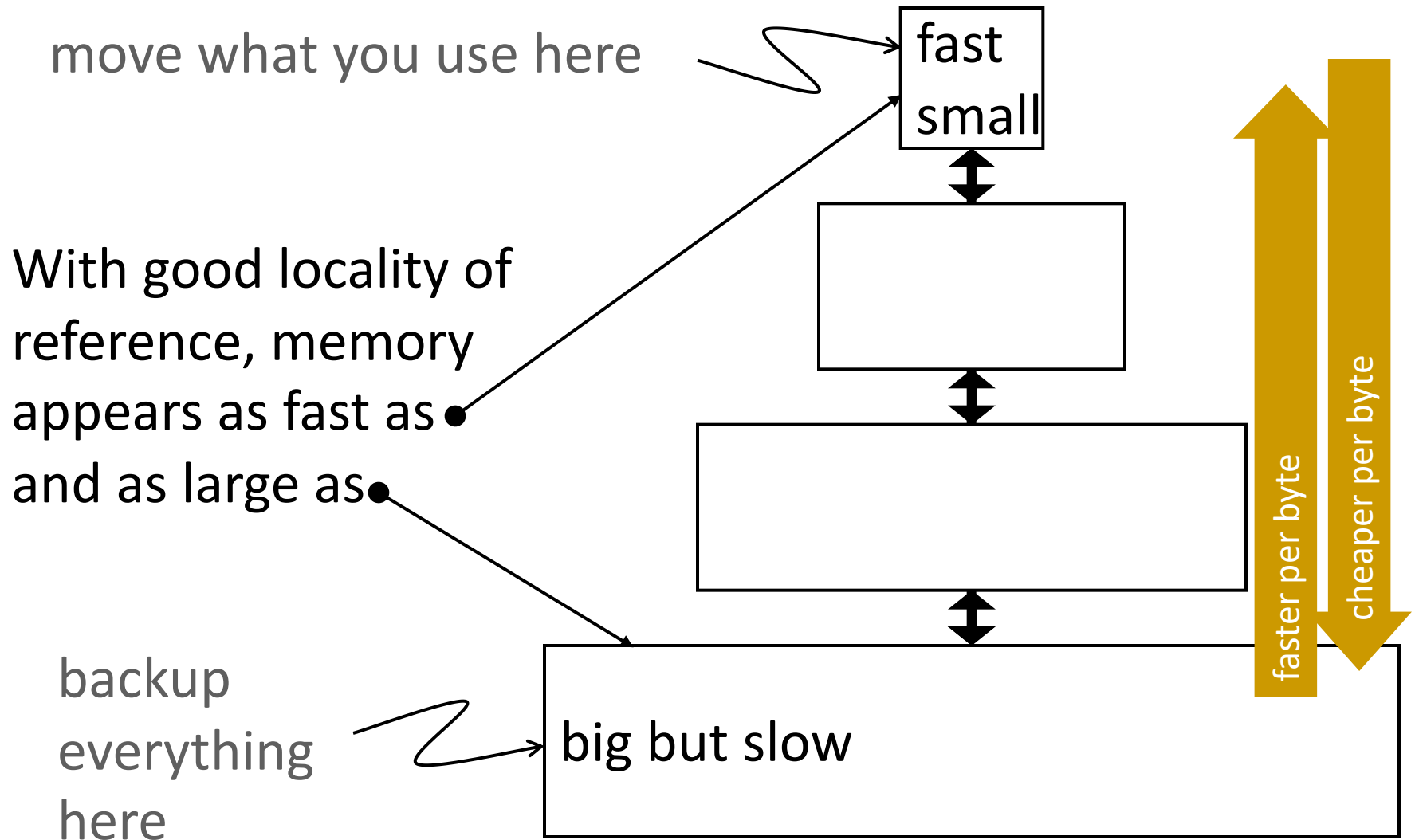
Memory Hierarchy

- **Fundamental tradeoff**
  - Fast memory: small
  - Large memory: slow
- Idea: Memory hierarchy

```
+-------+   +-------+          +----------+   +----------+
| CPU   |<->| Cache |<-------->| Main     |   |          |
| +---+ |   |       |          | Memory   |   | Hard Disk|
| |RF | |   |       |          | (DRAM)   |   |          |
| +---+ |   |       |          |          |   |          |
+-------+   +-------+          +----------+   +----------+
```
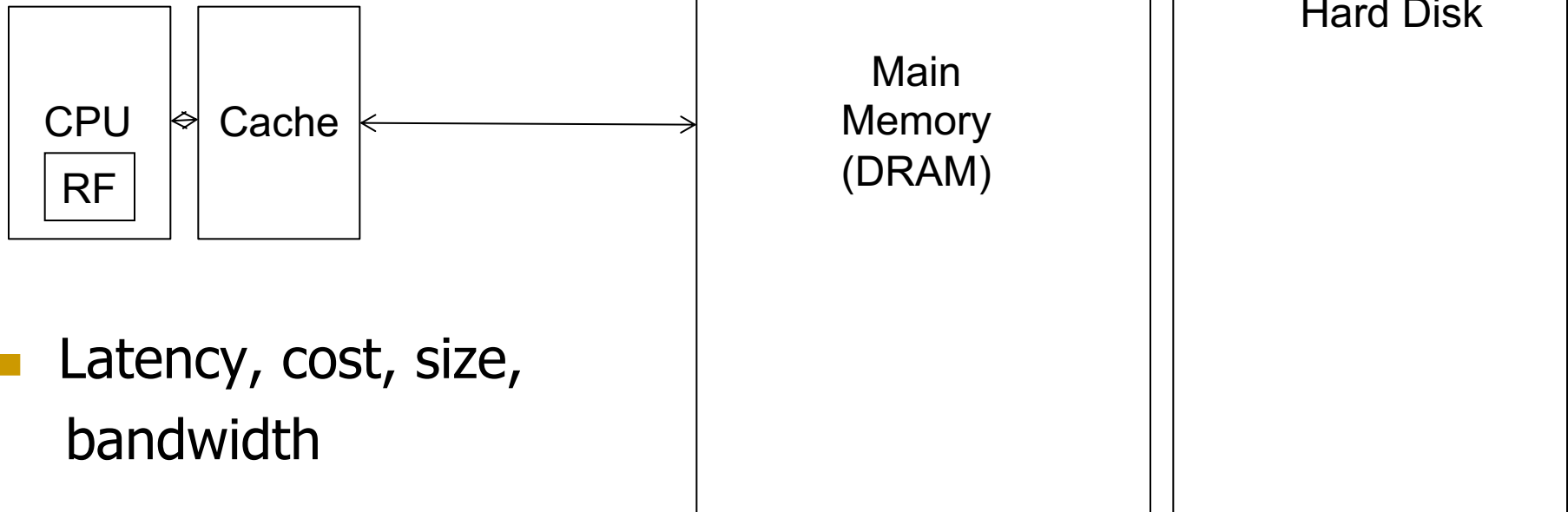
- **Latency, cost, size, bandwidth**

Locality

- One's recent past is a very good predictor of his/her near future.

- Temporal Locality: If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly

- Spatial Locality: If you did something, it is very likely you will do something similar/related (in space)
  - every time I find you in this room, you are probably sitting close to the same people

Memory Locality

- A "typical" program has a lot of locality in memory references
  - typical programs are composed of "loops"

- Temporal: A program tends to reference the same memory location many times and all within a small window of time

- Spatial: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    - 1. instruction memory references
    - 2. array/data structure references

Caching Basics: Exploit Temporal Locality

- Idea: Store recently accessed data in automatically managed fast memory (called cache)

- Anticipation: the data will be accessed again soon

- Temporal locality principle
  - Recently accessed data will be again accessed in the near future
  - This is what Maurice Wilkes had in mind:
    - Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.
    - "The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory."

Caching Basics: Exploit Spatial Locality

- Idea: Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- Anticipation: nearby data will be accessed soon

- Spatial locality principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
    - Liptay, "Structural aspects of the System/360 Model 85 II: the cache," IBM Systems Journal, 1968.
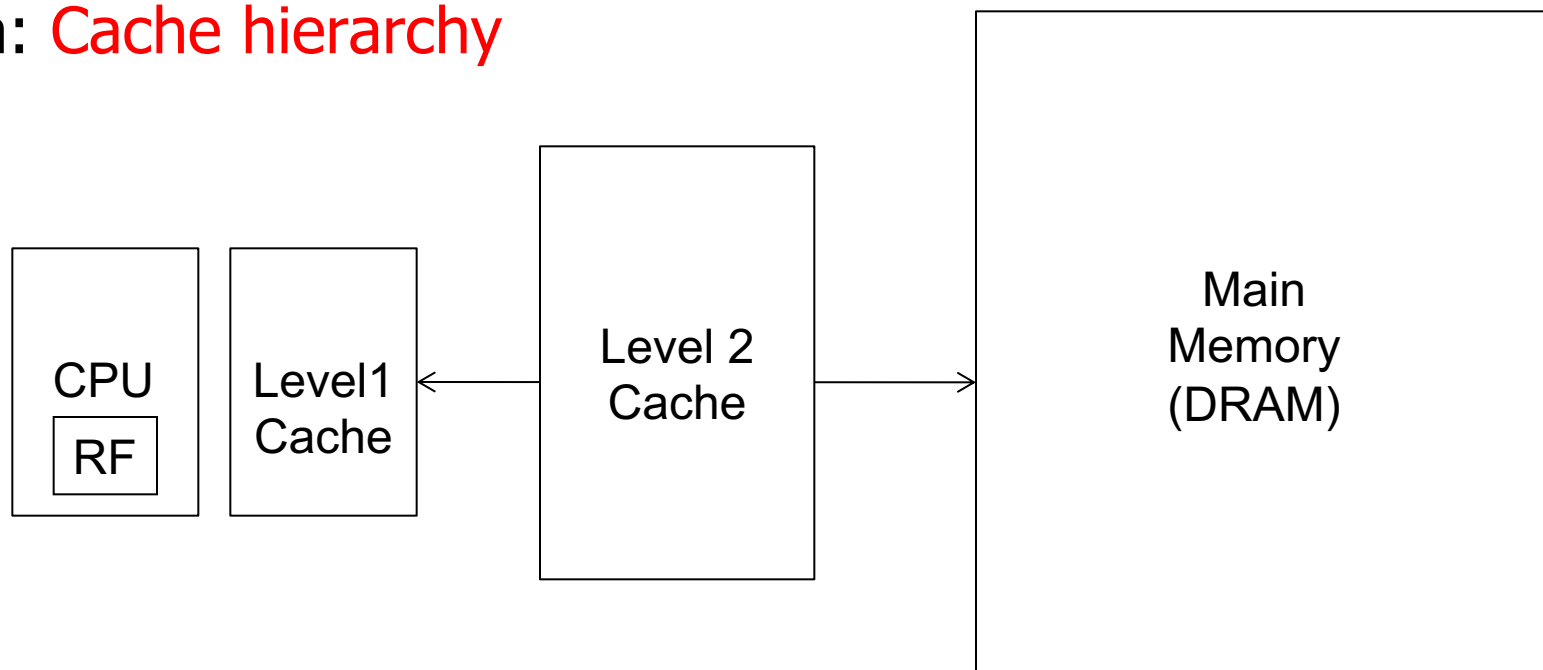
The Bookshelf Analogy

- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage

- Recently-used books tend to stay on desk
  - Comp Arch books, books for classes you are currently taking
  - Until the desk gets full
- Adjacent books in the shelf needed around the same time
  - If I have organized/categorized my books well in the shelf

Caching in a Pipelined Design

- The cache needs to be tightly integrated into the pipeline
  - Ideally, access in 1-cycle so that load-dependent operations do not stall
- High frequency pipeline → Cannot make the cache large
  - But, we want a large cache AND a pipelined design
- Idea: Cache hierarchy

```
┌─────────┐  ┌─────────┐     ┌─────────┐       ┌─────────────┐
│ CPU     │  │ Level1  │ ←── │ Level 2 │ ────→ │ Main        │
│ ┌─────┐ │  │ Cache   │     │ Cache   │       │ Memory      │
│ │ RF  │ │  │         │     │         │       │ (DRAM)      │
│ └─────┘ │  │         │     │         │       │             │
└─────────┘  └─────────┘     └─────────┘       └─────────────┘
```

A Note on Manual vs. Automatic Management

- **Manual:** Programmer manages data movement across levels

  -- too painful for programmers on substantial programs

  ❑ "core" vs "drum" memory in the 50's

  ❑ still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache) and GPUs (called "shared memory")

- **Automatic:** Hardware manages data movement across levels, transparently to the programmer

  ++ programmer's life is easier

  ❑ the average programmer doesn't need to know about it

    ▪ You don't need to know how big the cache is and how it works to write a "correct" program! (What if you want a "fast" program?)

Automatic Management in Memory Hierarchy

- Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Trans. On Electronic Computers, 1965.

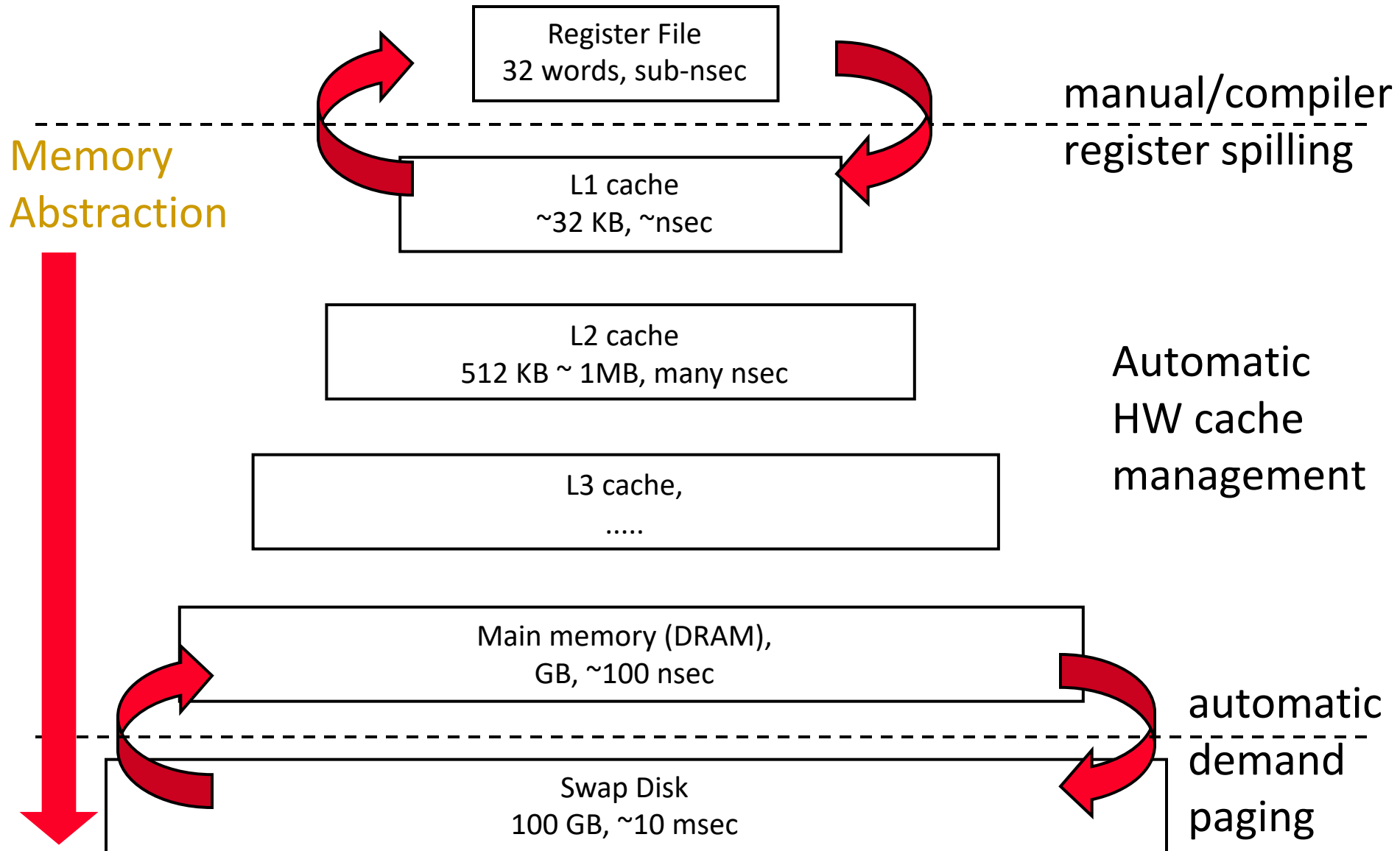## Slave Memories and Dynamic Storage Allocation

M. V. WILKES

SUMMARY

The use is discussed of a fast core memory of, say, 32 000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

- "By a slave memory I mean one which automatically accumulates to itself words that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again."

Historical Aside: Other Cache Papers

- Fotheringham, "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store," CACM 1961.

  - http://dl.acm.org/citation.cfm?id=366800

- Bloom, Cohen, Porter, "Considerations in the Design of a Computer with High Logic-to-Memory Speed Ratio," AIEE Gigacycle Computing Systems Winter Meeting, Jan. 1962.

A Modern Memory Hierarchy

Register File
32 words, sub-nsec

manual/compiler
register spilling

Memory
Abstraction

L1 cache
~32 KB, ~nsec

L2 cache
512 KB ~ 1MB, many nsec

Automatic
HW cache
management

L3 cache,
.....

Main memory (DRAM),
GB, ~100 nsec

automatic
demand
paging

Swap Disk
100 GB, ~10 msec

Hierarchical Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of t_i, The perceived access time T_i is longer than t_i

- Except for the outer-most hierarchy, when looking for a given address there is

    - a chance (hit-rate h_i) you "hit" and access time is t_i

    - a chance (miss-rate m_i) you "miss" and access time $t_i + T_{i+1}$

    - $h_i + m_i = 1$

- Thus

    $$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$
    $$T_i = t_i + m_i \cdot T_{i+1}$$

h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at L_{i-1}

Hierarchy Design Considerations

- Recursive latency equation

$$T_i = t_i + m_i \cdot T_{i+1}$$

- The goal: achieve desired T_1 within allowed cost

- $T_i \approx t_i$ is desirable

- Keep m_i low
  - increasing capacity C_i lowers m_i, but beware of increasing t_i
  - lower m_i by smarter cache management (replacement::anticipate what you don't need, prefetching::anticipate what you will need)

- Keep T_{i+1} low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise

Intel Pentium 4 Example

- **90nm P4, 3.6 GHz**
- **L1 D-cache**
  - C_1 = 16K
  - t_1 = 4 cyc int / 9 cycle fp
- **L2 D-cache**
  - C_2 = 1024 KB
  - t_2 = 18 cyc int / 18 cyc fp
- **Main memory**
  - t_3 = ~ 50ns or 180 cyc
- **Notice**
  - best case latency is not 1
  - worst case access latencies are into 500+ cycles

if m_1=0.1, m_2=0.1
T_1=7.6, T_2=36

if m_1=0.01, m_2=0.01
T_1=4.2, T_2=19.8

if m_1=0.05, m_2=0.01
T_1=5.00, T_2=19.8

if m_1=0.01, m_2=0.50
T_1=5.08, T_2=108

Cache Basics and Operation

Cache

- Generically, any structure that "memoizes" frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g. a web cache

- Most commonly in the processor design context: an automatically-managed memory structure based on SRAM

  - memoize in SRAM the most frequently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency

Caching Basics

- **Block (line):** Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache

- On a reference:
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - Maybe have to kick something else out to do it

- Some important cache design decisions
  - Placement: where and how to place/find a block in cache?
  - Replacement: what data to remove to make room in cache?
  - Granularity of management: large or small blocks? Subblocks?
  - Write policy: what do we do about writes?
  - Instructions/data: do we treat them separately?

Cache Abstraction and Metrics
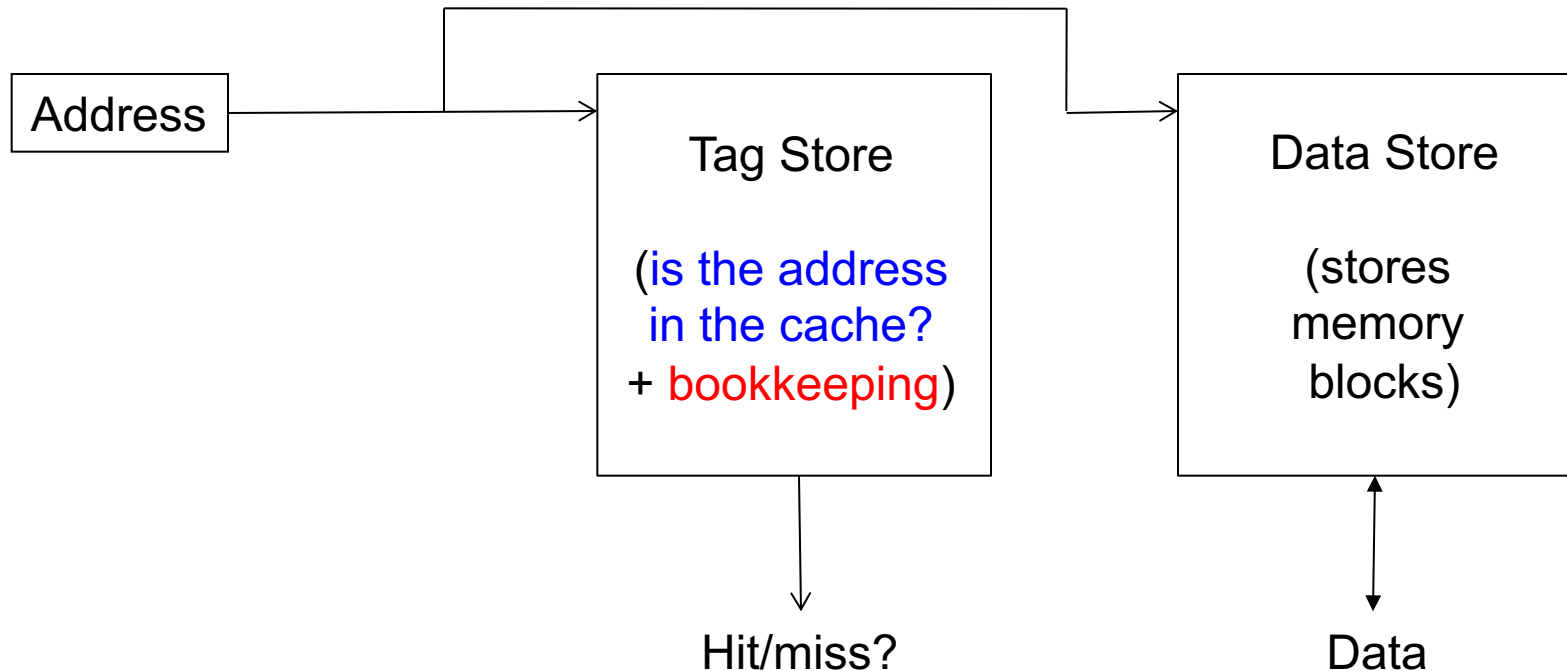
- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)

   = (hit-rate * hit-latency) + (miss-rate * miss-latency)
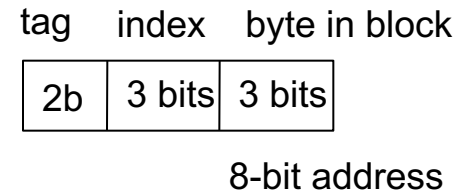- Aside: *Is reducing AMAT always beneficial for performance?*

A Basic Hardware Cache Design

- We will start with a basic hardware cache design

- Then, we will examine a multitude of ideas to make it better

Blocks and Addressing the Cache

- Memory is logically divided into fixed-size blocks

- Each block maps to a location in the cache, determined by the index bits in the address
  - used to index into the tag and data stores

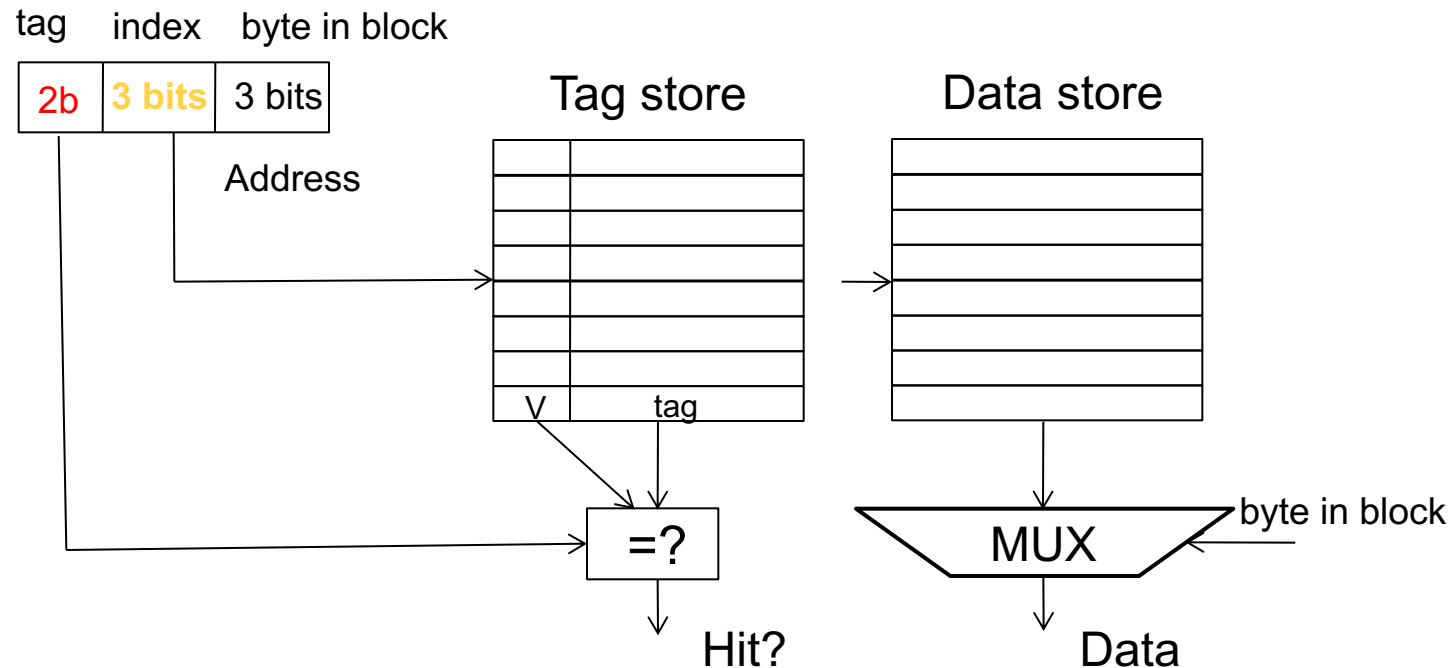| tag	index	byte in block
2b	3 bits	3 bits

8-bit address

- Cache access:

  1) index into the tag and data stores with index bits in address

  2) check valid bit in tag store

  3) compare tag bits in address with the stored tag in tag store

- If a block is in the cache (cache hit), the stored tag should be valid and match the tag of the block

Direct-Mapped Cache: Placement and Access

Block: **00**000
Block: 00001
Block: 00010
Block: 00011
Block: 00100
Block: 00101
Block: 00110
Block: 00111
Block: **01**000
Block: 01001
Block: 01010
Block: 01011
Block: 01100
Block: 01101
Block: 01110
Block: 01111
Block: **10**000
Block: 10001
Block: 10010
Block: 10011
Block: 10100
Block: 10101
Block: 10110
Block: 10111
Block: **11**000
Block: 11001
Block: 11010
Block: 11011
Block: 11100
Block: 11101
Block: 11110
Block: 11111

Main memory

- Assume byte-addressable memory: 256 bytes, 8-byte blocks → 32 blocks

- Assume cache: 64 bytes, 8 blocks

  - Direct-mapped: A block can go to only one location

tag   index   byte in block

| 2b | 3 bits | 3 bits |

Address

Tag store

Data store

V   tag

=?

MUX ← byte in block

Hit?   Data

  - Addresses with same index contend for the same location

    - Cause conflict misses

Direct-Mapped Caches

- Direct-mapped cache: Two blocks in memory that map to the same index in the cache cannot be present in the cache at the same time
  - One index → one entry

- Can lead to 0% hit rate if more than one block accessed in an interleaved manner map to the same index
  - Assume addresses A and B have the same index bits but different tag bits
  - A, B, A, B, A, B, A, B, … → conflict in the cache index
  - All accesses are conflict misses

Set Associativity

- Addresses 0 and 8 always conflict in direct mapped cache
- Instead of having one column of 8, have 2 columns of 4 blocks

Tag store

Data store

SET

V tag   V tag

=?   =?

Logic

Hit?

MUX

MUX   byte in block

Address

tag   index   byte in block

3b   2 bits   3 bits

Key idea: Associative memory within the set
+ Accommodates conflicts better (fewer conflict misses)
-- More complex, slower access, larger tag store

Higher Associativity

- 4-way

Tag store

=? =? =? =?

Logic → Hit?

Data store

MUX

MUX ← byte in block

+ Likelihood of conflict misses even lower

-- More tag comparators and wider data mux; larger tags

Full Associativity

- Fully associative cache
  - A block can be placed in any cache location

Tag store [=?] [=?] [=?] [=?] [=?] [=?] [=?] [=?]

Logic

Hit?

Data store

MUX

MUX — byte in block

Associativity (and Tradeoffs)

- **Degree of associativity**: How many blocks can map to the same index (or set)?

- Higher associativity

  ++ Higher hit rate

  -- Slower cache access time (hit latency and data access latency)

  -- More expensive hardware (more comparators)

- Diminishing returns from higher associativity

hit rate

associativity

Issues in Set-Associative Caches

- Think of each block in a set having a "priority"
  - Indicating how important it is to keep the block in the cache
- Key issue: How do you determine/adjust block priorities?
- There are three key decisions in a set:
  - Insertion, promotion, eviction (replacement)

- Insertion: What happens to priorities on a cache fill?
  - Where to insert the incoming block, whether or not to insert the block
- Promotion: What happens to priorities on a cache hit?
  - Whether and how to change block priority
- Eviction/replacement: What happens to priorities on a cache miss?
  - Which block to evict and how to adjust priorities

Eviction/Replacement Policy

- **Which block** in the set **to replace** on a cache miss?
  - ❑ Any invalid block first
  - ❑ If all are valid, consult the **replacement policy**
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - Least costly to re-fetch?
      - ❑ Why would memory accesses have different cost?
    - Hybrid replacement policies
    - Optimal replacement policy?

Implementing LRU

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks

- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly?

- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly?
  - How many different orderings possible for the 4 blocks in the set?
  - How many bits needed to encode the LRU order of a block?
  - What is the logic needed to determine the LRU victim?

Approximations of LRU

- Most modern processors do not implement "true LRU" (also called "perfect LRU") in highly-associative caches

- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)

- Examples:
  - Not MRU (not most recently used)
  - Hierarchical LRU: divide the N-way set into M "groups", track the MRU group and the MRU way in each group
  - Victim-NextVictim Replacement: Only keep track of the victim and the next victim

Cache Replacement Policy: LRU or Random

- LRU vs. Random: Which one is better?
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy

- Set thrashing: When the "program working set" in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs

- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar

- Best of both Worlds: Hybrid of LRU and Random
  - How to choose between the two? Set sampling
    - See Qureshi et al., "A Case for MLP-Aware Cache Replacement," ISCA 2006.

What Is the Optimal Replacement Policy?

- Belady's OPT
  - Replace the block that is going to be referenced furthest in the future by the program
  - Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, 1966.
  - How do we implement this? Simulate?

- Is this optimal for minimizing miss rate?
- Is this optimal for minimizing execution time?
  - No. Cache miss latency/cost varies from block to block!
  - Two reasons: Remote vs. local caches and miss overlapping
  - Qureshi et al. "A Case for MLP-Aware Cache Replacement," ISCA 2006.

Reading

- Key observation: Some misses more costly than others as their latency is exposed as stall time. Reducing miss rate is not always good for performance. Cache replacement should take into account MLP of misses.

- Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt,
  "A Case for MLP-Aware Cache Replacement"
  Proceedings of the 33rd International Symposium on Computer Architecture (**ISCA**), pages 167-177, Boston, MA, June 2006. Slides (ppt)

A Case for MLP-Aware Cache Replacement

Moinuddin K. Qureshi   Daniel N. Lynch   Onur Mutlu   Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{moin, lynch, onur, patt}@hps.utexas.edu

Digital Design & Computer Arch.

Lecture 21b: Memory Hierarchy and Caches

Prof. Onur Mutlu

ETH Zürich

Spring 2020

14 May 2020

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

What's In A Tag Store Entry?

- Valid bit

- Tag

- Replacement policy bits

- Dirty bit?
  - Write back vs. write through caches

Handling Writes (I)

- **When do we write the modified data in a cache to the next level?**
    - **Write through**: At the time the write happens
    - **Write back**: When the block is evicted

- **Write-back**
    + Can combine multiple writes to the same block before eviction
        - Potentially saves bandwidth between cache levels + saves energy
    -- Need a bit in the tag store indicating the block is "dirty/modified"

- **Write-through**
    + Simpler
    + All levels are up to date. **Consistency**: Simpler cache coherence because no need to check close-to-processor caches' tag stores for presence
    -- More bandwidth intensive; no combining of writes

Handling Writes (II)

- **Do we allocate a cache block on a write miss?**
  - **Allocate on write miss:** Yes
  - **No-allocate on write miss:** No

- Allocate on write miss
  - \+ Can combine writes instead of writing each of them individually to next level
  - \+ Simpler because write misses can be treated the same way as read misses
  - \-\- Requires (?) transfer of the whole cache block

- No-allocate
  - \+ Conserves cache space if locality of writes is low (potentially better cache hit rate)

Handling Writes (III)

- What if the processor writes to an entire block over a small amount of time?

- Is there any need to bring the block into the cache from memory in the first place?

- Ditto for a *portion* of the block, i.e., subblock
  - E.g., 4 bytes out of 64 bytes

Sectored Caches

- Idea: Divide a block into subblocks (or sectors)
  - Have separate valid and dirty bits for each sector
  - When is this useful? (Think writes…)

++ No need to transfer the entire cache block into the cache
   (A write simply validates and updates a subblock)

++ More freedom in transferring subblocks into the cache (a cache block does not need to be in the cache fully)
   (How many subblocks do you transfer on a read?)

-- More complex design

-- May not exploit spatial locality fully when used for reads

v	d	subblock	v	d	subblock	● ● ● ●	v	d	subblock	tag

Instruction vs. Data Caches

- Separate or Unified?

- Unified:

    + Dynamic sharing of cache space: no overprovisioning that might happen with static partitioning (i.e., split I and D caches)

    -- Instructions and data can thrash each other (i.e., no guaranteed space for either)

    -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

- First level caches are almost always split

    - Mainly for the last reason above

- Second and higher levels are almost always unified

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity
  - Tag store and data store accessed in parallel
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store accessed serially

- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter (filters some temporal and spatial locality)
    - Management policies are therefore different

Cache Performance

Cache Parameters vs. Miss/Hit Rate

- Cache size

- Block size

- Associativity

- Replacement policy
- Insertion/Placement policy

Cache Size

- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- Too large a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- Too small a cache
  - doesn't exploit temporal locality well
  - useful data replaced often

- Working set: the whole set of data the executing application references
  - Within a time interval

hit rate

"working set" size

cache size
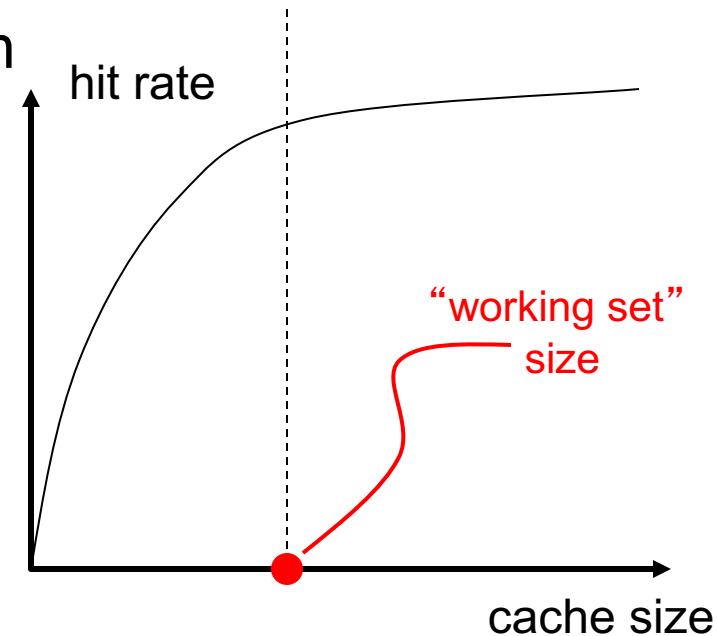
Block Size

- Block size is the data that is associated with an address tag
  - not necessarily the unit of transfer between hierarchies
    - Sub-blocking: A block divided into multiple pieces (each with V bit)
      - Can improve "write" performance

- Too small blocks
  - don't exploit spatial locality well
  - have larger tag overhead

- Too large blocks
  - too few total # of blocks → less temporal locality exploitation
  - waste of cache space and bandwidth/energy if spatial locality is not high

hit rate

block size
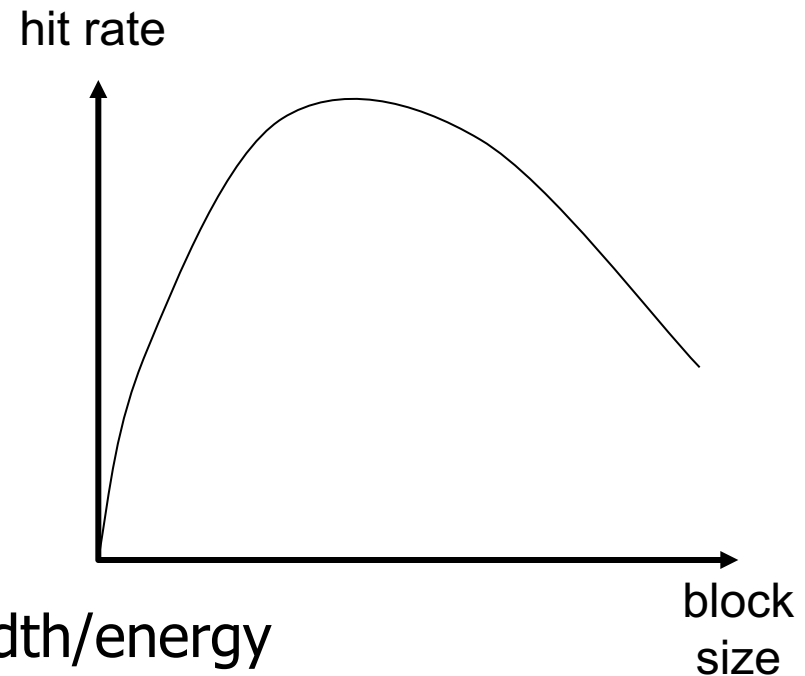
Large Blocks: Critical-Word and Subblocking

- Large cache blocks can take a long time to fill into the cache
  - fill cache line critical word first
  - restart cache access before complete fill

- Large cache blocks can waste bus bandwidth
  - divide a block into subblocks
  - associate separate valid bits for each subblock
  - When is this useful?

| v | d | subblock | v | d | subblock | ● ● ● ● | v | d | subblock | tag |

Associativity

- How many blocks can be present in the same index (i.e., set)?

- Larger associativity
  - lower miss rate (reduced conflicts)
  - higher hit latency and area cost (plus diminishing returns)

- Smaller associativity
  - lower cost
  - lower hit latency
    - Especially important for L1 caches

- Is power of 2 associativity required?

hit rate

associativity
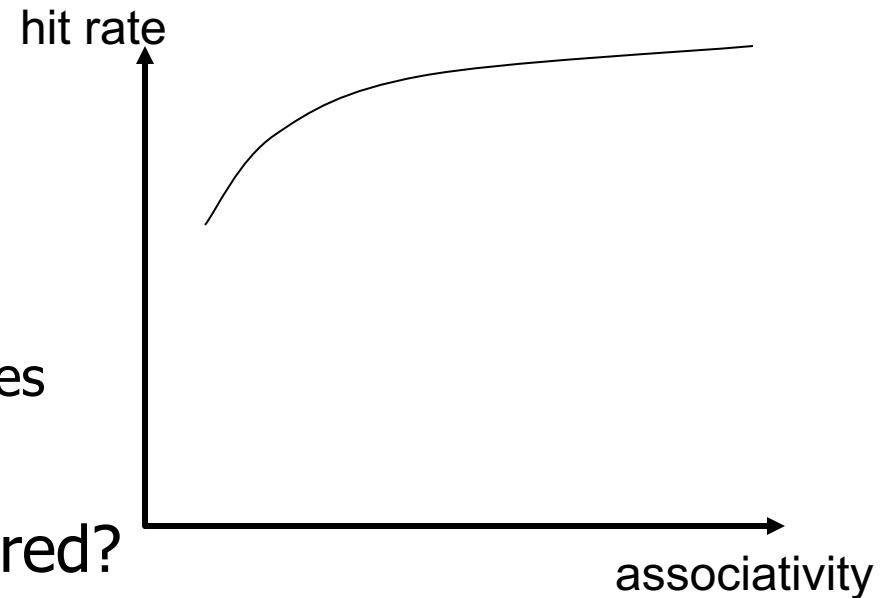
Classification of Cache Misses

- **Compulsory miss**
  - first reference to an address (block) always results in a miss
  - subsequent references should hit unless the cache block is displaced for the reasons below

- **Capacity miss**
  - cache is too small to hold everything needed
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity

- **Conflict miss**
  - defined as any miss that is neither a compulsory nor a capacity miss

How to Reduce Each Miss Type

- Compulsory
  - Caching cannot help
  - Prefetching can
- Conflict
  - More associativity
  - Other ways to get more associativity without making the cache associative
    - Victim cache
    - Better, randomized indexing
    - Software hints?
- Capacity
  - Utilize cache space better: keep blocks that will be referenced
  - Software management: divide working set such that each "phase" fits in cache

How to Improve Cache Performance

- Three fundamental goals

- Reducing miss rate
  - Caveat: reducing miss rate can reduce performance if more costly-to-refetch blocks are evicted

- Reducing miss latency or miss cost

- Reducing hit latency or hit cost

- The above three **together** affect performance

Cache Examples:
For You to Study

Cache Terminology

- **Capacity (C):**
  - the number of data bytes a cache stores

- **Block size (b):**
  - bytes of data brought into cache at once

- **Number of blocks ($B = C/b$):**
  - number of blocks in cache: $B = C/b$

- **Degree of associativity (N):**
  - number of blocks in a set

- **Number of sets ($S = B/N$):**
  - each memory address maps to exactly one cache set

How is data found?

- **Cache organized into *S* sets**

- **Each memory address maps to exactly one set**

- **Caches categorized by number of blocks in a set:**
  - *Direct mapped:* 1 block per set
  - *N-way set associative:* N blocks per set
  - *Fully associative:* all cache blocks are in a single set

- **Examine each organization for a cache with:**
  - Capacity (*C* = 8 words)
  - Block size (*b* = 1 word)
  - So, number of blocks (*B* = 8)

Direct Mapped Cache

Address

Address	
11...111**111**00	mem[0xFF...FC]
11...111**110**00	mem[0xFF...F8]
11...111**101**00	mem[0xFF...F4]
11...111**100**00	mem[0xFF...F0]
11...111**011**00	mem[0xFF...EC]
11...111**010**00	mem[0xFF...E8]
11...111**001**00	mem[0xFF...E4]
11...111**000**00	mem[0xFF...E0]
⋮	⋮
00...001**001**00	mem[0x00...24]
00...001**000**00	mem[0x00..20]
00...000**111**00	mem[0x00..1C]
00...000**110**00	mem[0x00...18]
00...000**101**00	mem[0x00...14]
00...000**100**00	mem[0x00...10]
00...000**011**00	mem[0x00...0C]
00...000**010**00	mem[0x00...08]
00...000**001**00	mem[0x00...04]
00...000**000**00	mem[0x00...00]

2^{30} Word Main Memory

Set Number

	Set Number
	7 (**111**)
	6 (**110**)
	5 (**101**)
	4 (**100**)
	3 (**011**)
	2 (**010**)
	1 (**001**)
	0 (**000**)

2^3 Word Cache

Direct Mapped Cache Hardware
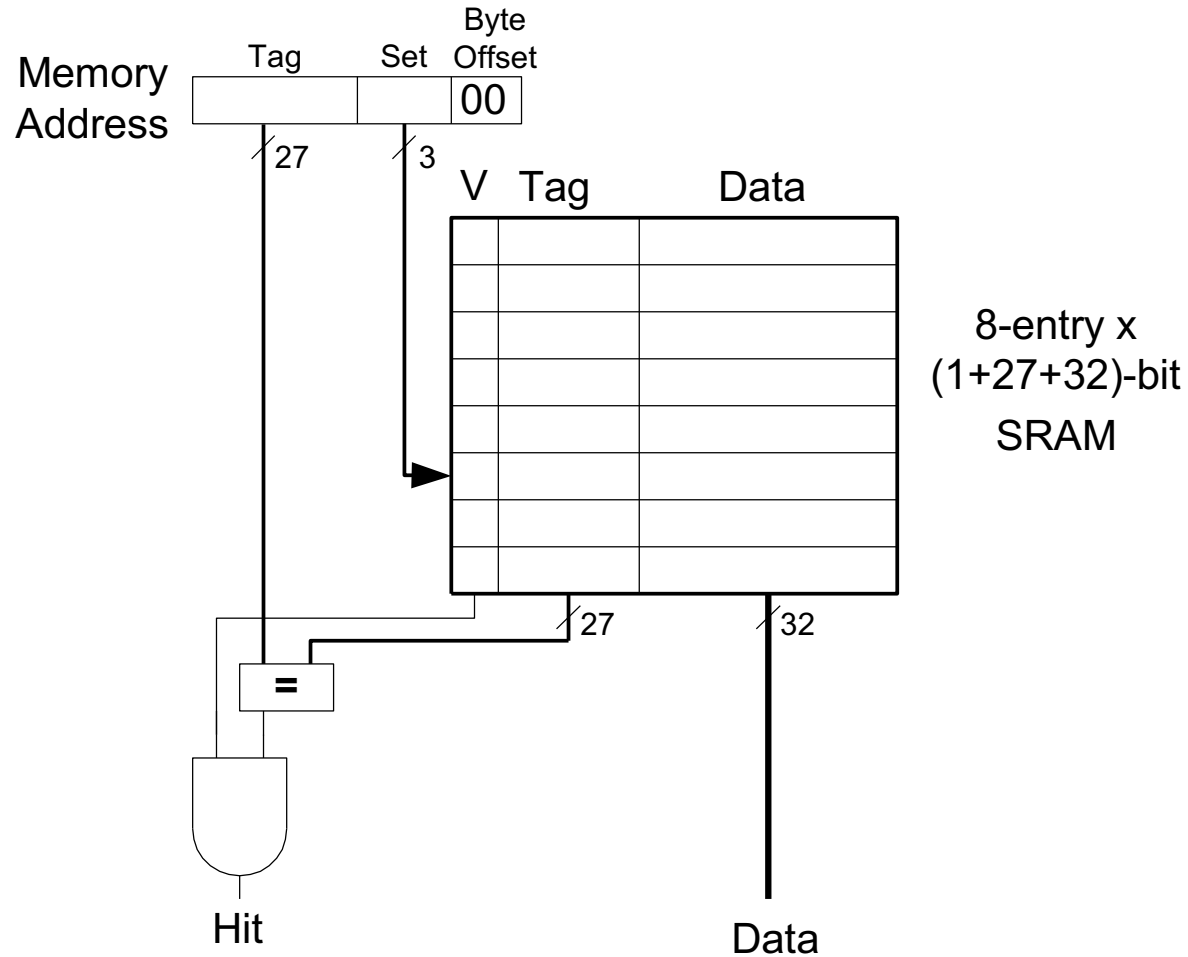
Direct Mapped Cache Performance
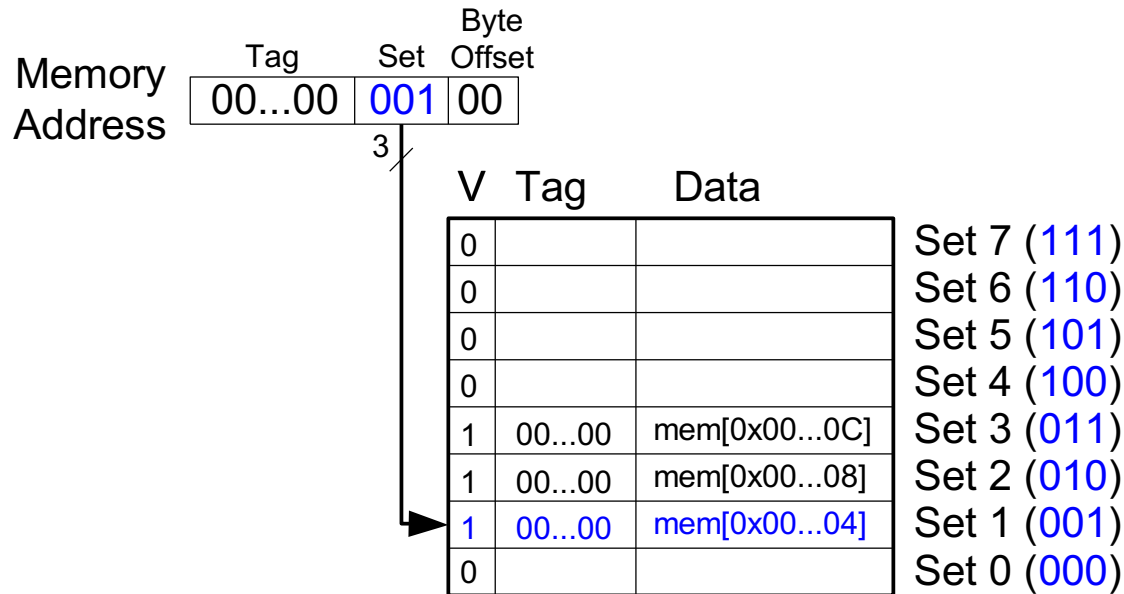
Memory Address

	Tag	Set	Byte Offset
	00...00	001	00

3

V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
1	00...00	mem[0x00...0C]	Set 3 (011)
1	00...00	mem[0x00...08]	Set 2 (010)
1	00...00	mem[0x00...04]	Set 1 (001)
0			Set 0 (000)

```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate   =

Direct Mapped Cache Performance

Memory Address

	Tag	Set	Byte Offset
	00...00	001	00

3

V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
1	00...00	mem[0x00...0C]	Set 3 (011)
1	00...00	mem[0x00...08]	Set 2 (010)
1	00...00	mem[0x00...04]	Set 1 (001)
0			Set 0 (000)

```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```
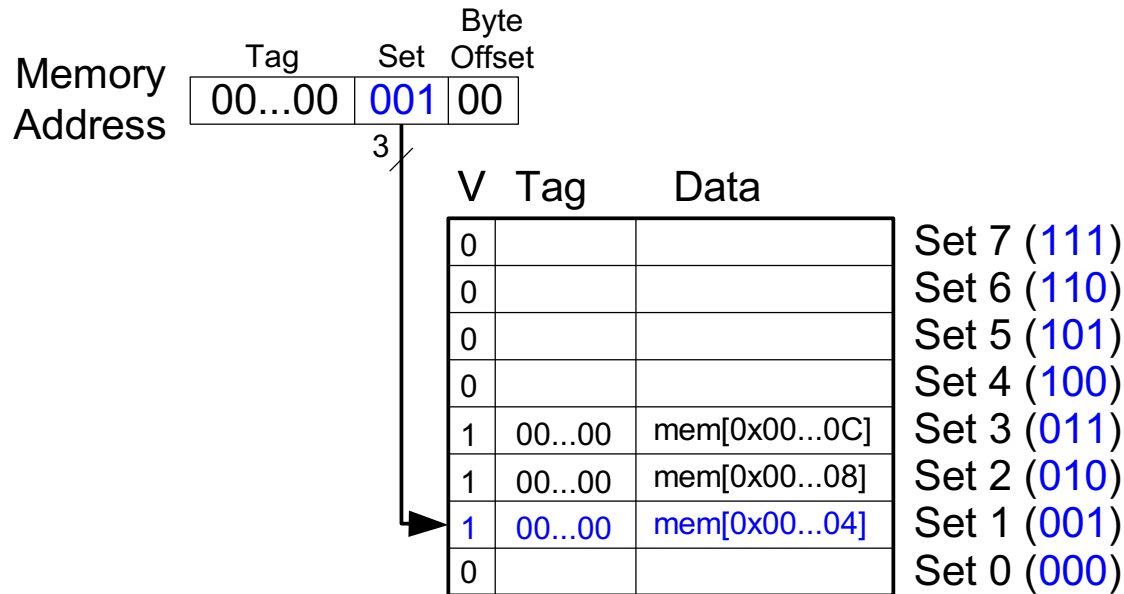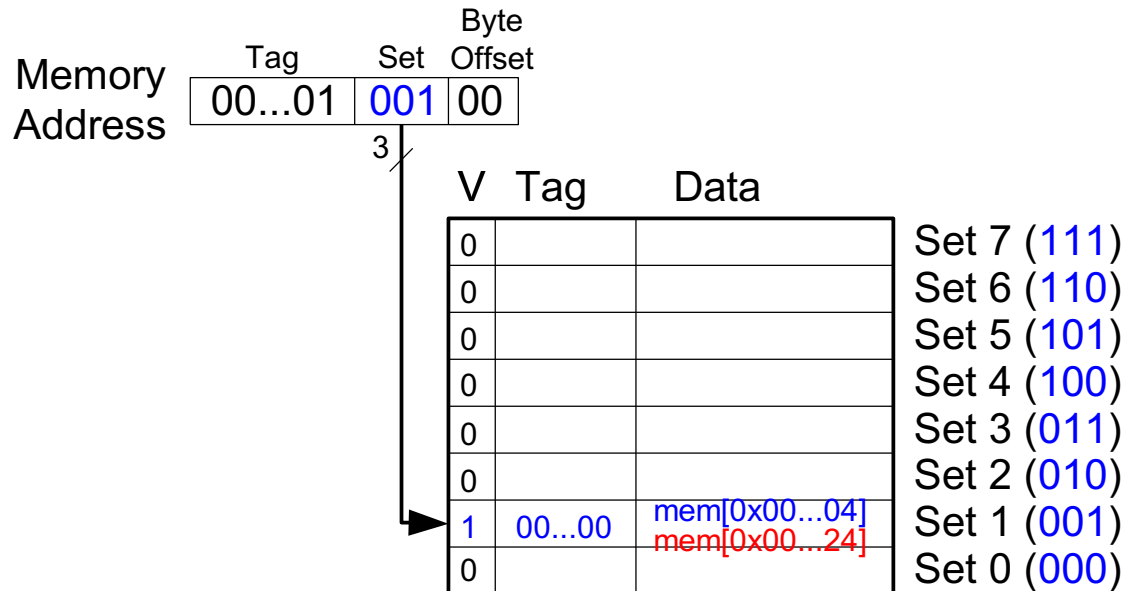
Miss Rate   = 3/15

= 20%

Temporal Locality

Compulsory Misses

70

Direct Mapped Cache: Conflict

Memory Address

Tag	Set	Byte Offset
00...01	001	00

3

V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
0			Set 3 (011)
0			Set 2 (010)
1	00...00	mem[0x00...04] mem[0x00...24]	Set 1 (001)
0			Set 0 (000)

```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =

Direct Mapped Cache: Conflict

Memory Address

Tag	Set	Byte Offset
00...01	001	00

3

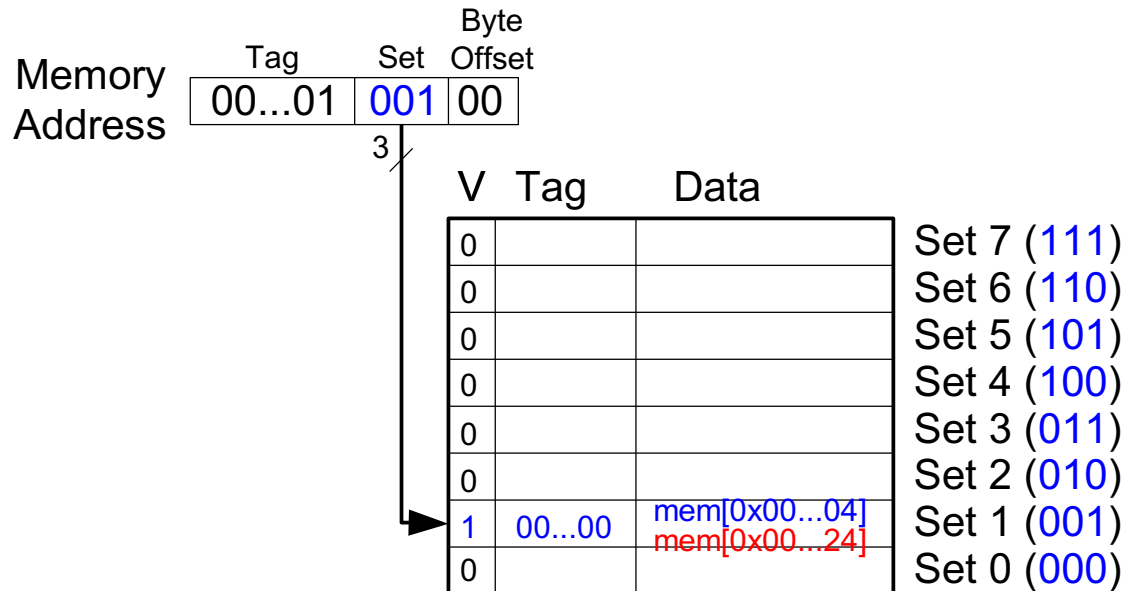| V	Tag	Data	
0			Set 7 (111)
0			Set 6 (110)
0			Set 5 (101)
0			Set 4 (100)
0			Set 3 (011)
0			Set 2 (010)
1	00...00	mem[0x00...04] mem[0x00...24]	Set 1 (001)
0			Set 0 (000)

```
# MIPS assembly code
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate   = 10/10
         = 100%

Conflict Misses

N-Way Set Associative Cache

N-way Set Associative Performance
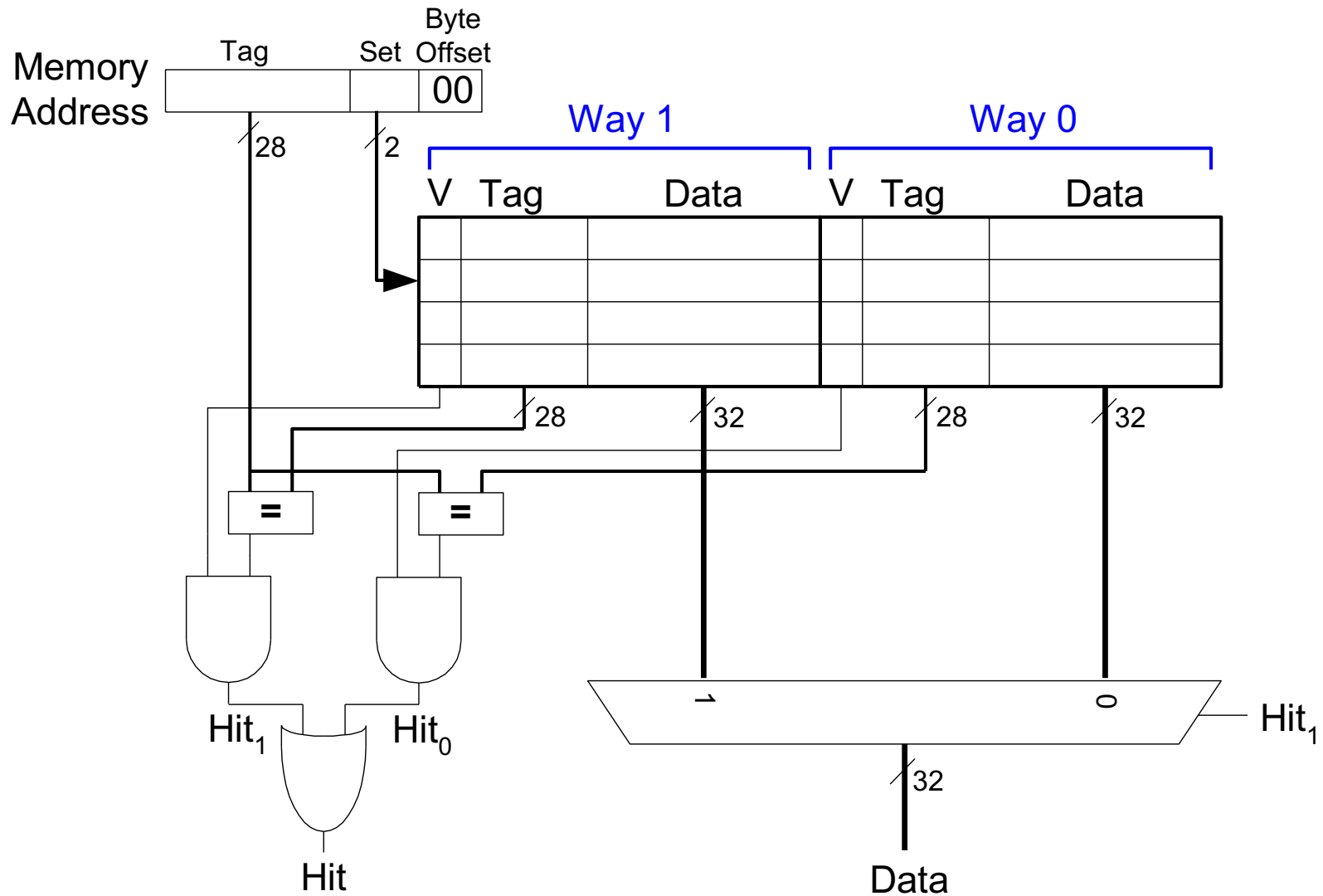
```
# MIPS assembly code

        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0x24($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =

	Way 1				Way 0		
V	Tag	Data		V	Tag	Data	
0				0			Set 3
0				0			Set 2
1	00...10	mem[0x00...24]		1	00...00	mem[0x00...04]	Set 1
0				0			Set 0

N-way Set Associative Performance

```
# MIPS assembly code

        addi  $t0, $0, 5
loop:   beq   $t0, $0, done
        lw    $t1, 0x4($0)
        lw    $t2, 0x24($0)
        addi  $t0, $t0, -1
        j     loop
done:
```

Miss Rate = 2/10

= 20%

Associativity reduces conflict misses

Way 1

Way 0

V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 0

Fully Associative Cache

- **No conflict misses**

- **Expensive to build**

V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data

Spatial Locality?

- **Increase block size:**
  - Block size, *b = 4* words
  - *C* = 8 words
  - Direct mapped (1 block per set)
  - Number of blocks, $B = C/b = 8/4 = 2$
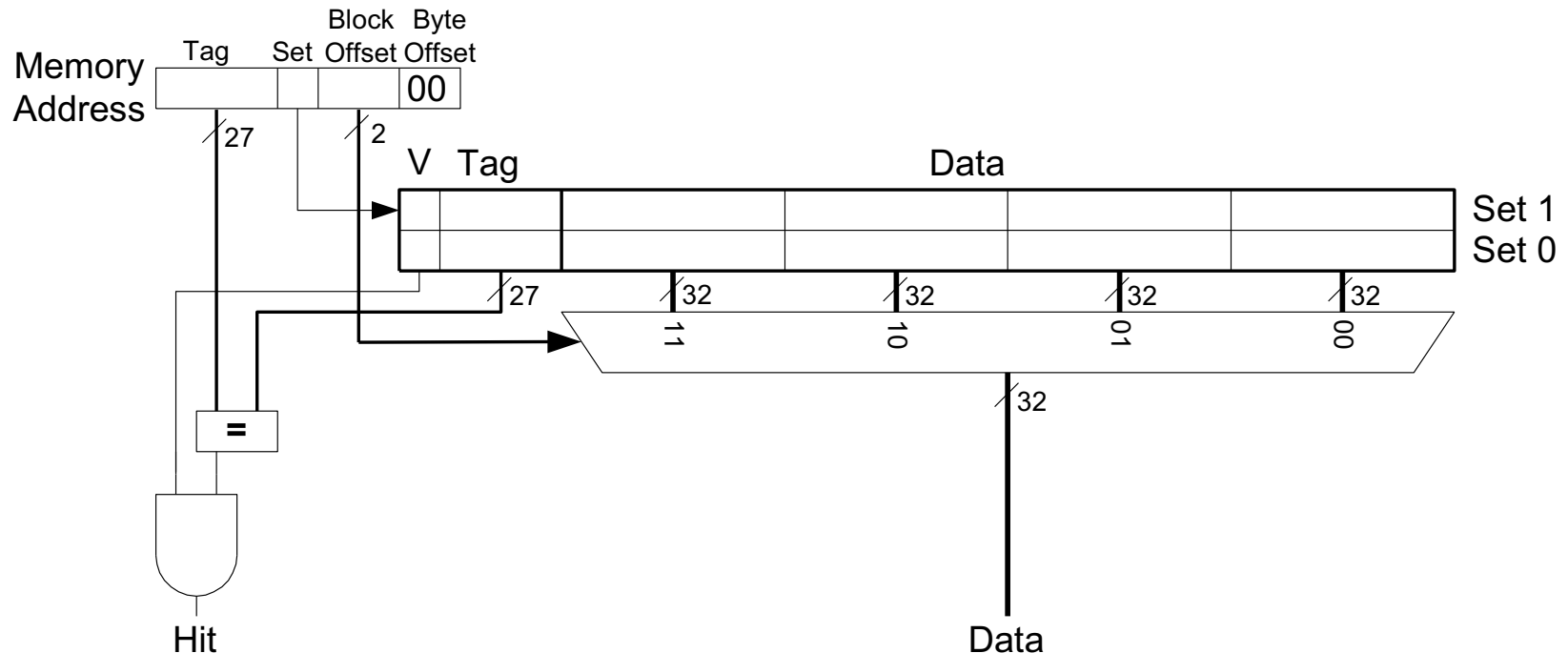
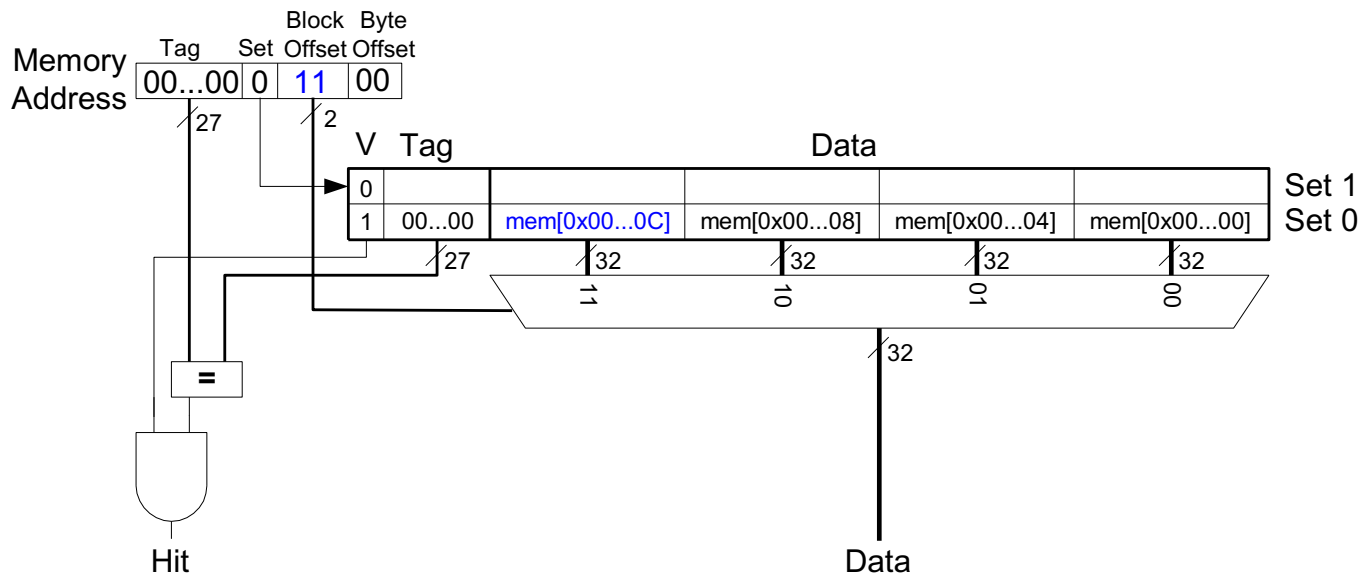# Direct Mapped Cache Performance

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```

Miss Rate =

Direct Mapped Cache Performance

```
        addi $t0, $0, 5
loop:   beq  $t0, $0, done
        lw   $t1, 0x4($0)
        lw   $t2, 0xC($0)
        lw   $t3, 0x8($0)
        addi $t0, $t0, -1
        j    loop
done:
```
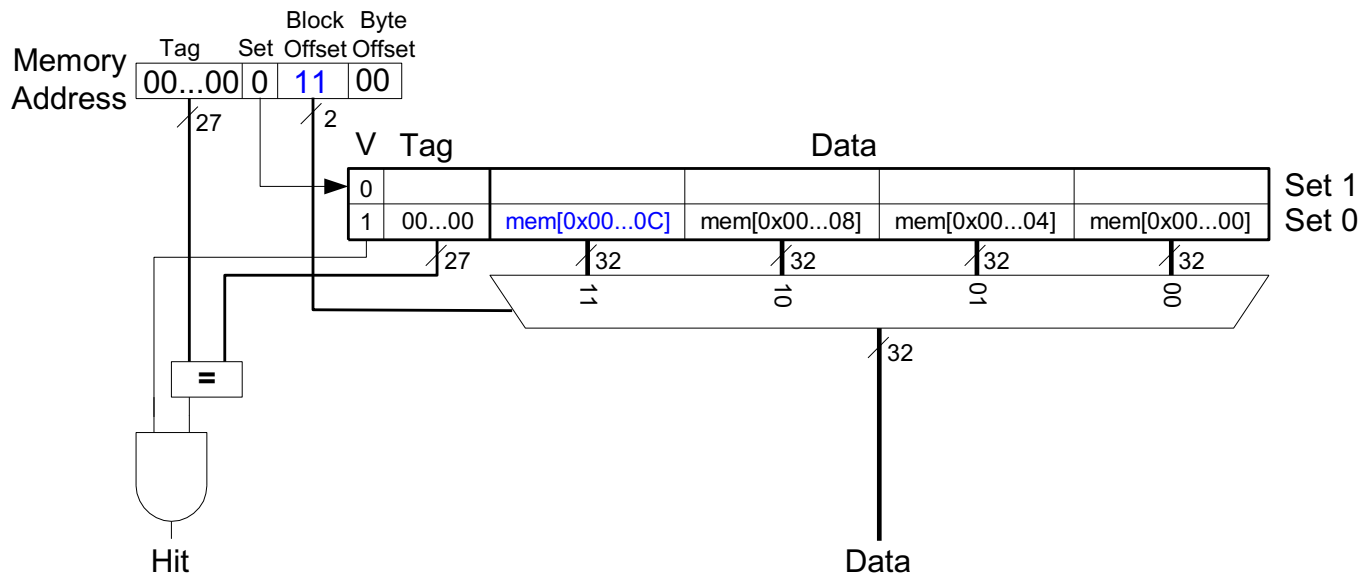
Miss Rate = 1/15

= 6.67%

Larger blocks reduce compulsory misses through spatial locality

Cache Organization Recap

- **Main Parameters**
  - Capacity: *C*
  - Block size: *b*
  - Number of blocks in cache: *B* = *C/b*
  - Number of blocks in a set: *N*
  - Number of Sets: *S* = *B/N*

Organization	Number of Ways (N)	Number of Sets (S = B/N)
Direct Mapped	1	B
N-Way Set Associative	1 < N < B	B / N
Fully Associative	B	1

Capacity Misses

- **Cache is too small to hold all data of interest at one time**
    - If the cache is full and program tries to access data X that is not in cache, cache must evict data Y to make room for X
    - *Capacity miss* occurs if program then tries to access Y again
    - X will be placed in a particular set based on its address

- **In a *direct mapped* cache, there is only one place to put X**

- **In an *associative cache*, there are multiple ways where X could go in the set.**

- **How to choose Y to minimize chance of needing it again?**
    - Least recently used (LRU) replacement: the least recently used block in a set is evicted when the cache is full.

Types of Misses

- *Compulsory:* **first time data is accessed**

- *Capacity:* **cache too small to hold all data of interest**

- *Conflict:* **data of interest maps to same location in cache**

- *Miss penalty:* **time it takes to retrieve a block from lower level of hierarchy**

LRU Replacement

	V	U	Tag	Data	V	Tag	Data	Set Number
(a)								3 (**11**)
								2 (**10**)
								1 (**01**)
								0 (**00**)

	V	U	Tag	Data	V	Tag	Data	Set Number
(b)								3 (**11**)
								2 (**10**)
								1 (**01**)
								0 (**00**)

LRU Replacement

```
# MIPS assembly

lw $t0, 0x04($0)
lw $t1, 0x24($0)
lw $t2, 0x54($0)
```

(a)

		Way 1			Way 0		
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]	Set 1 (01)
0	0			0			Set 0 (00)

(b)

		Way 1			Way 0		
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]	Set 1 (01)
0	0			0			Set 0 (00)